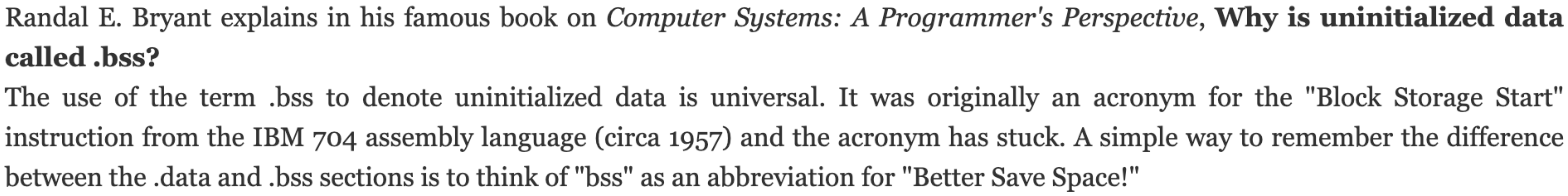
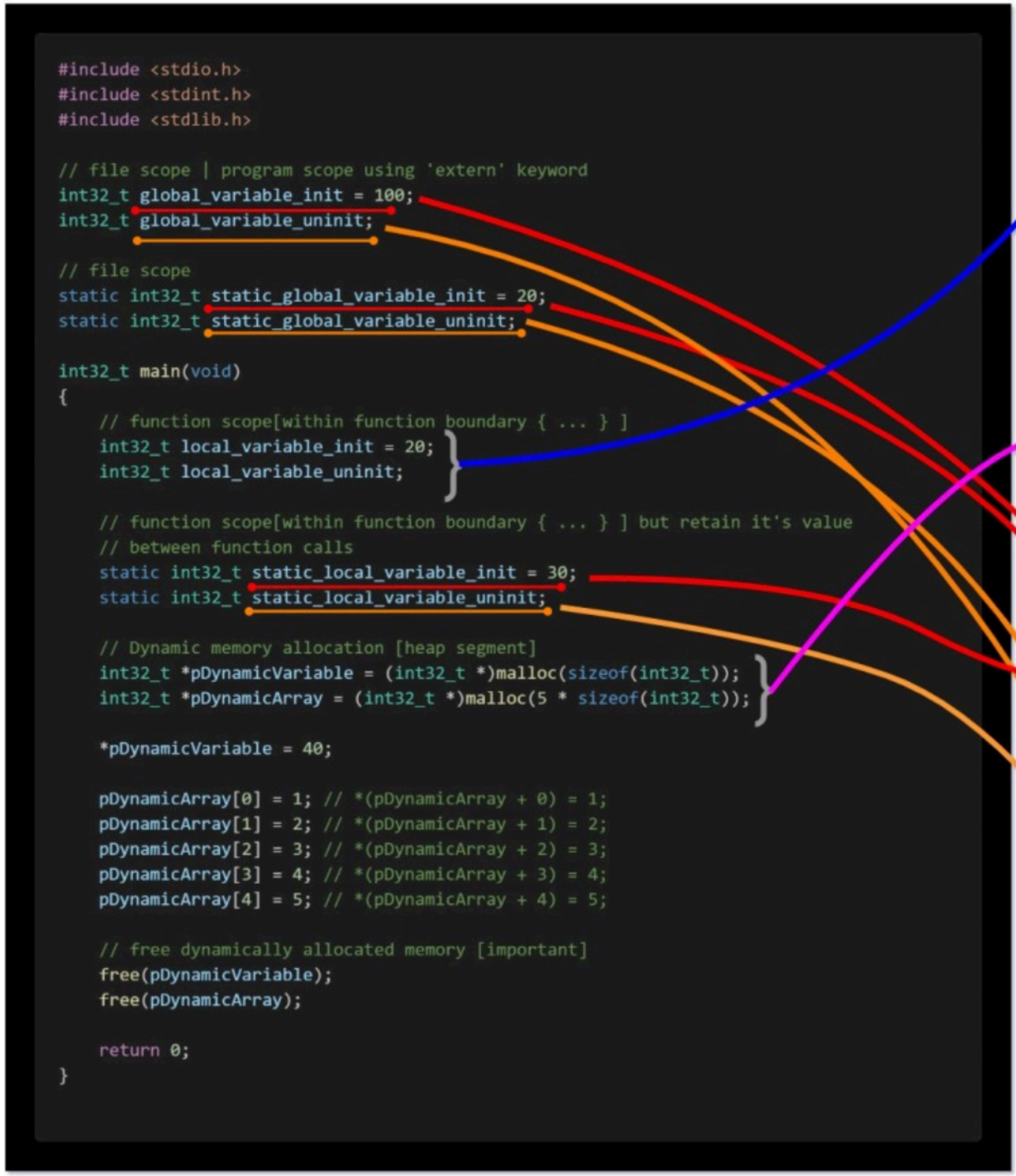
https://cs-fundamentals.com/c-programming/memory-layout-of-c-program-code-data-segments

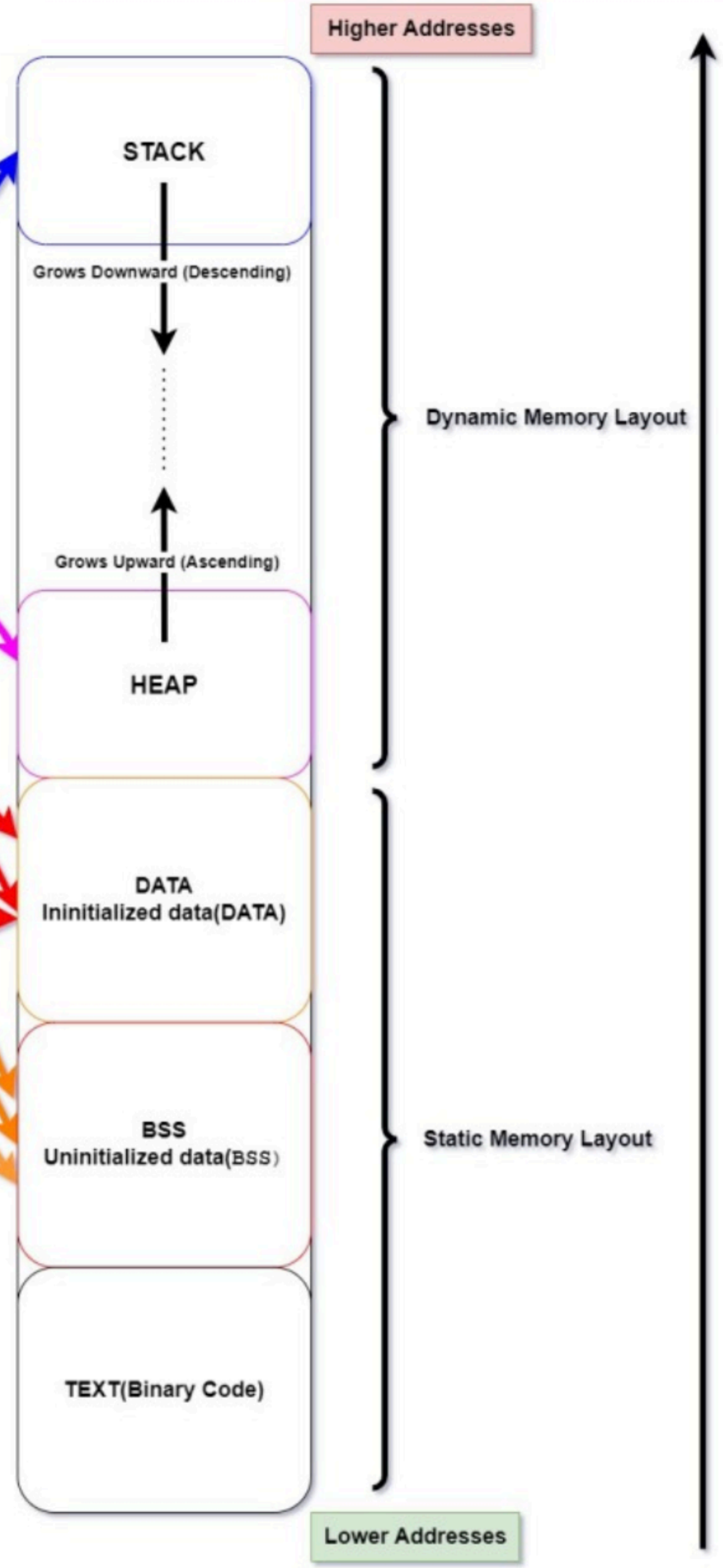


Randal E. Bryant explains in his famous book on *Computer Systems: A Programmer's Perspective*, **Why is uninitialized data called .bss?**

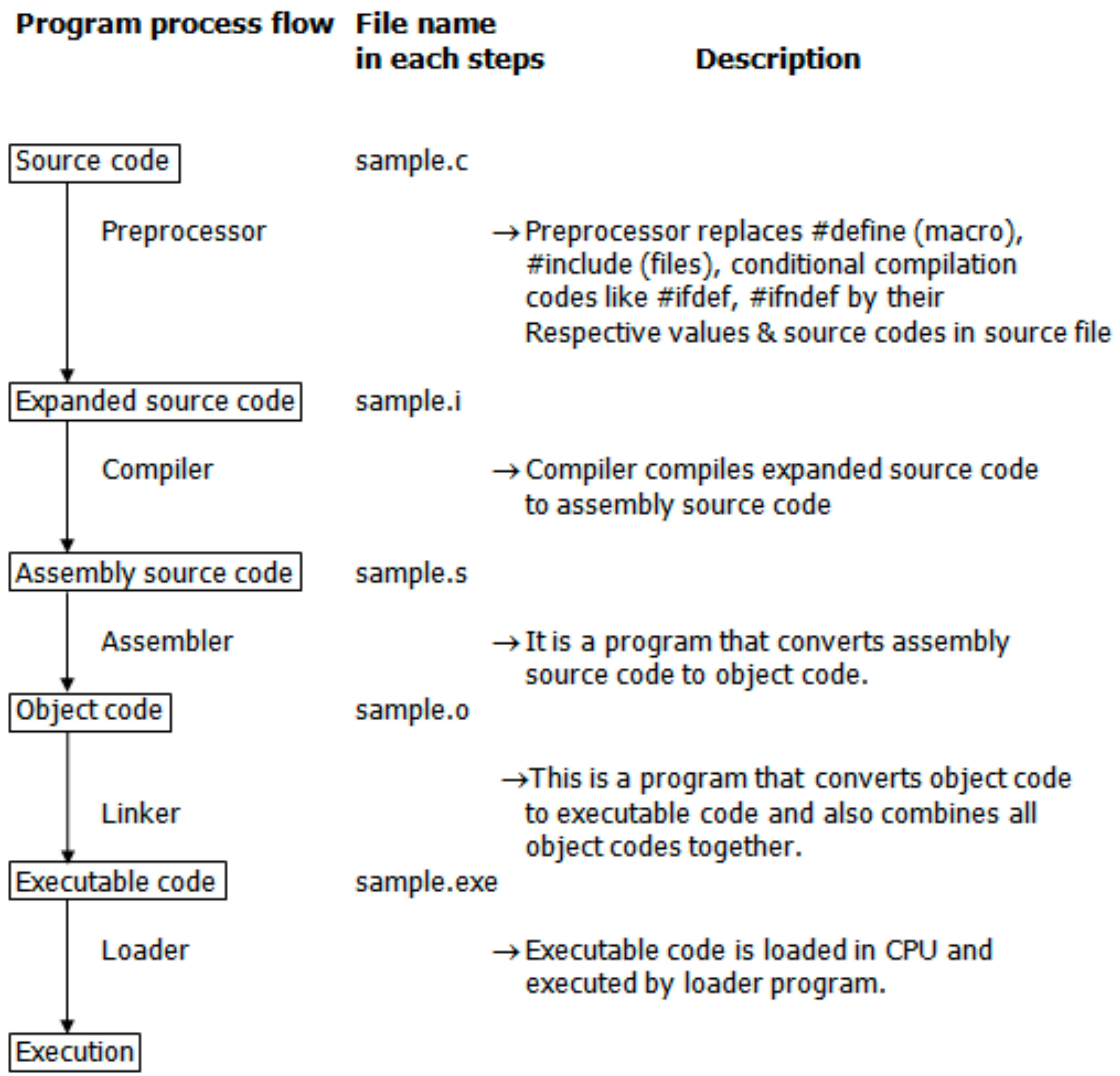
The use of the term .bss to denote uninitialized data is universal. It was originally an acronym for the "Block Storage Start" instruction from the IBM 704 assembly language (circa 1957) and the acronym has stuck. A simple way to remember the difference between the .data and .bss sections is to think of "bss" as an abbreviation for "Better Save Space!"



```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

// file scope | program scope using 'extern' keyword
int32_t global_variable_init = 100;
int32_t global_variable_uninit;

// file scope
static int32_t static_global_variable_init = 20;
static int32_t static_global_variable_uninit;

int32_t main(void)
{
    // function scope[within function boundary { ... } ]
    int32_t local_variable_init = 20;
    int32_t local_variable_uninit;

    // function scope[within function boundary { ... } ] but retain it's value
    // between function calls
    static int32_t static_local_variable_init = 30;
    static int32_t static_local_variable_uninit;

    // Dynamic memory allocation [heap segment]
    int32_t *pDynamicVariable = (int32_t *)malloc(sizeof(int32_t));
    int32_t *pDynamicArray = (int32_t *)malloc(5 * sizeof(int32_t));

    *pDynamicVariable = 40;

    pDynamicArray[0] = 1; // *(pDynamicArray + 0) = 1;
    pDynamicArray[1] = 2; // *(pDynamicArray + 1) = 2;
    pDynamicArray[2] = 3; // *(pDynamicArray + 2) = 3;
    pDynamicArray[3] = 4; // *(pDynamicArray + 3) = 4;
    pDynamicArray[4] = 5; // *(pDynamicArray + 4) = 5;

    // free dynamically allocated memory [important]
    free(pDynamicVariable);
    free(pDynamicArray);

    return 0;
}
```



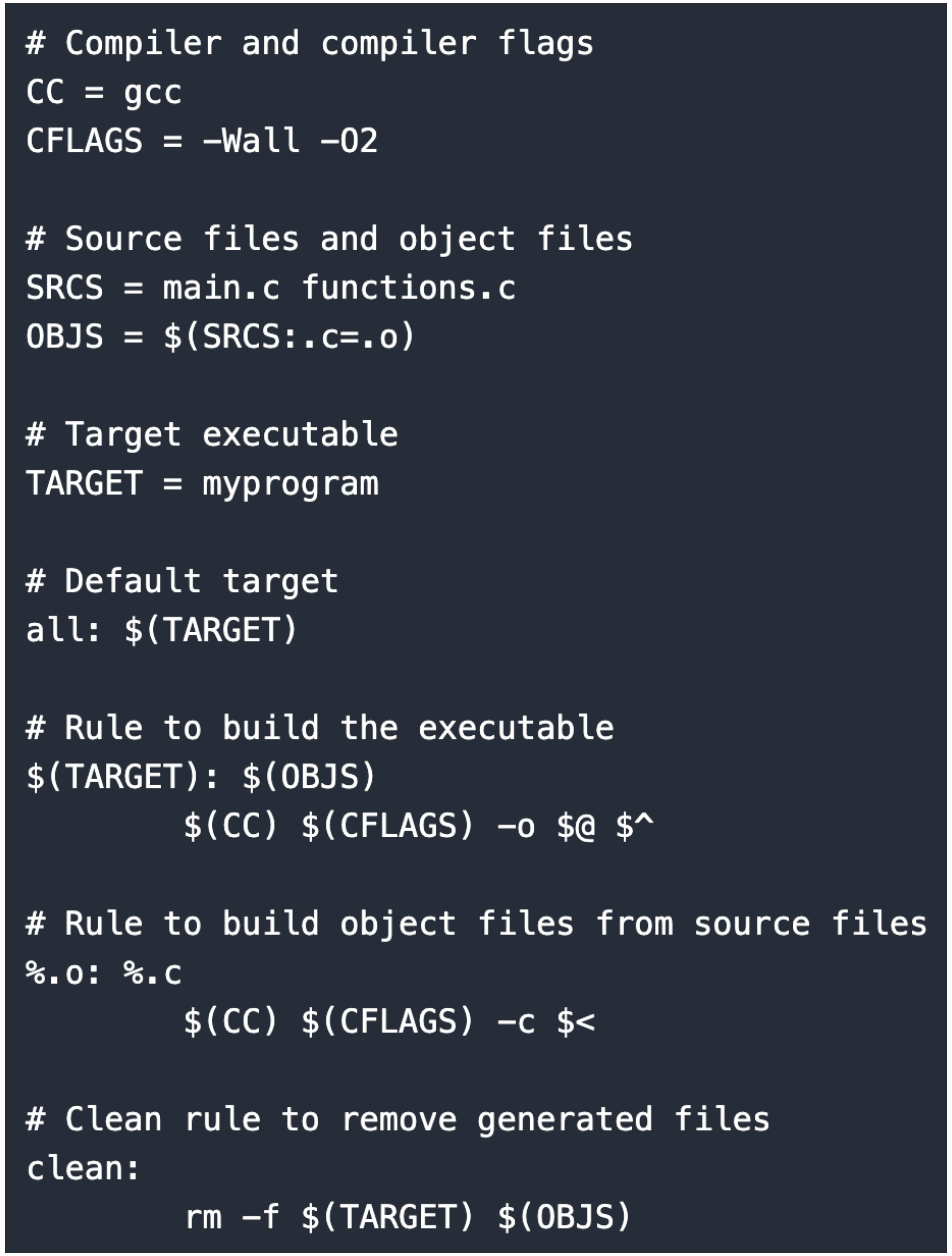
https://www.fresh2refresh.com/c-programming/c-preprocessor-directives/
https://2711.medium.com/from-preprocessing-to-linking-what-happens-under-the-hood-when-you-type-gcc-main-c-f61a246b89a0



Program process flow	File name in each steps	Description
Source code	sample.c	
Preprocessor		→ Preprocessor replaces #define (macro), #include (files), conditional compilation codes like #ifdef, #ifndef by their Respective values & source codes in source file
Expanded source code	sample.i	
Compiler		→ Compiler compiles expanded source code to assembly source code
Assembly source code	sample.s	
Assembler		→ It is a program that converts assembly source code to object code.
Object code	sample.o	
Linker		→ This is a program that converts object code to executable code and also combines all object codes together.
Executable code	sample.exe	
Loader		→ Executable code is loaded in CPU and executed by loader program.
Execution		

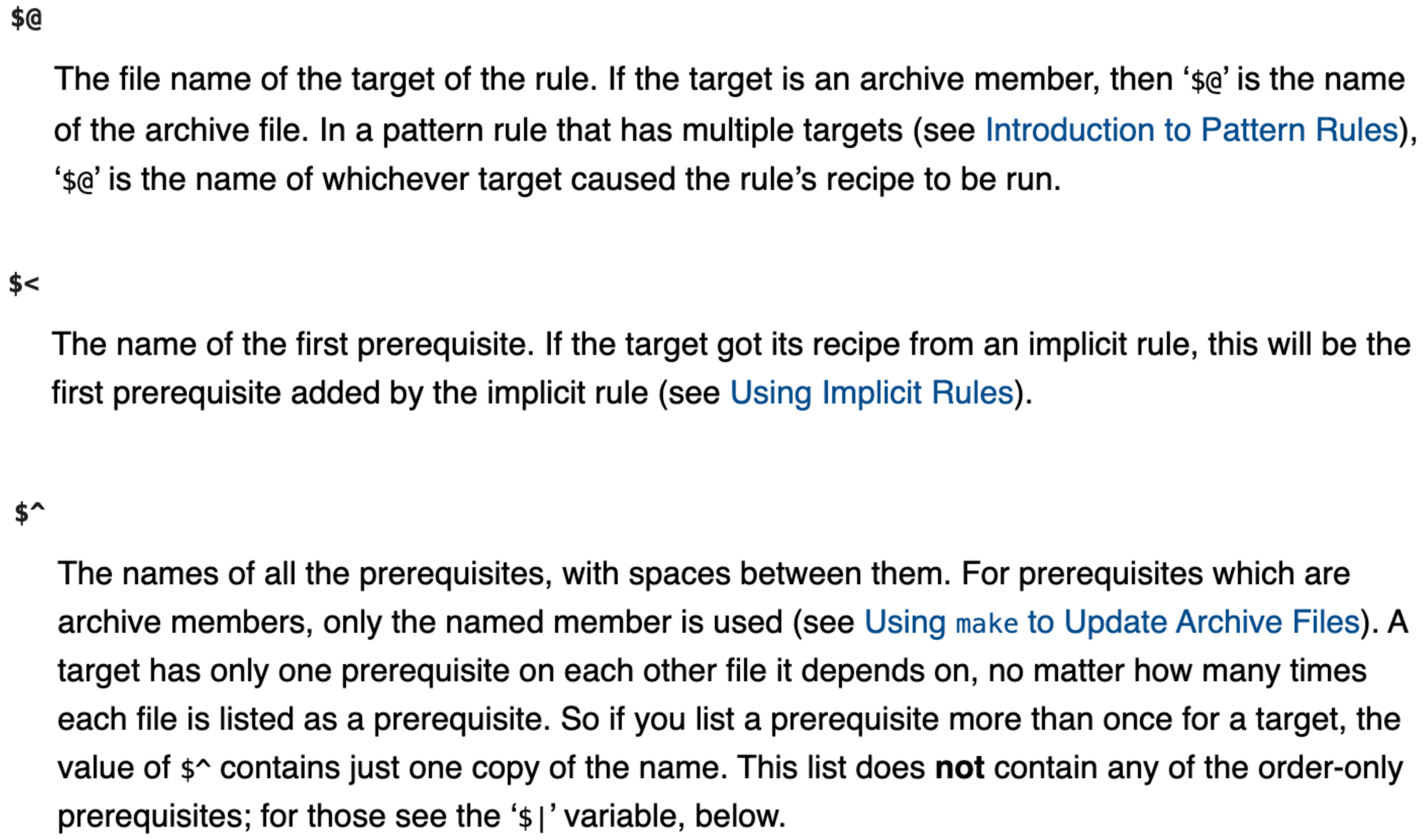
```
# This is a comment line
CC=g++
# CFLAGS will be the options passed to the compiler.
CFLAGS= -c  -Wall

all: prog

prog: main.o factorial.o hello.o
		$(CC) main.o factorial.o hello.o -o prog

main.o: main.cpp
		$(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
		$(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
		$(CC) $(CFLAGS) hello.cpp

clean:
		rm -rf   *.o
```

https://www.linkedin.com/pulse/makefile-uttam-basu/



```
# Compiler and compiler flags
CC = gcc
CFLAGS = -Wall -O2

# Source files and object files
SRCS = main.c functions.c
OBJS = $(SRCS:.c=.o)

# Target executable
TARGET = myprogram

# Default target
all: $(TARGET)

# Rule to build the executable
$(TARGET): $(OBJS)
        $(CC) $(CFLAGS) -o $@ $^

# Rule to build object files from source files
%.o: %.c
        $(CC) $(CFLAGS) -c $<

# Clean rule to remove generated files
clean:
        rm -f $(TARGET) $(OBJS)
```

https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables



`$@`

The file name of the target of the rule. If the target is an archive member, then '`$@`' is the name of the archive file. In a pattern rule that has multiple targets (see Introduction to Pattern Rules), '`$@`' is the name of whichever target caused the rule's recipe to be run.

`$<`

The name of the first prerequisite. If the target got its recipe from an implicit rule, this will be the first prerequisite added by the implicit rule (see Using Implicit Rules).

`$^`

The names of all the prerequisites, with spaces between them. For prerequisites which are archive members, only the named member is used (see Using `make` to Update Archive Files). A target has only one prerequisite on each other file it depends on, no matter how many times each file is listed as a prerequisite. So if you list a prerequisite more than once for a target, the value of `$^` contains just one copy of the name. This list does **not** contain any of the order-only prerequisites; for those see the '`$|`' variable, below.

5.3 String Length

You can get the length of a string using the `strlen` function. This function is declared in the header file `string.h`.

Function: *size_t* **strlen** *(const char *s)*

```c
/* strlen.c */

#include <stdio.h>
#include <string.h>

int main()
{
  char *t = "XXX";
  printf( "Length of <%s> is %d.\n", t, strlen( t ));
}
```

2

Function: *char* * **strcpy** *(char *restrict to, const char *restrict from)*

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See POSIX Safety Concepts.

This copies bytes from the string *from* (up to and including the terminating null byte) into the string *to*.

```
/* strcpy.c */

#include <stdio.h>
#include <string.h>

int main()
{
  char s1[100],
       s2[100];
  strcpy( s1, "xxxxxx 1" );
  strcpy( s2, "zzzzzz 2" );

  puts( "Original strings: " );
  puts( "" );
  puts( s1 );
  puts( s2 );
  puts( "" );

  strcpy( s2, s1 );

  puts( "New strings: " );
  puts( "" );
  puts( s1 );
  puts( s2 );
}
```

3

Function: *char* * **strcat** *(char *restrict to, const char *restrict from)*

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See POSIX Safety Concepts.

The `strcat` function is similar to `strcpy`, except that the bytes from *from* are concatenated or appended to the end of *to*, instead of overwriting it.

```c
/* strcat.c */

#include <stdio.h>
#include <string.h>

int main()
{
  char s1[50],
       s2[50];
  strcpy( s1, "Tweedledee " );
  strcpy( s2, "Tweedledum" );
  strcat( s1, s2 );
  puts( s1 );
}
```

4

```c
#include <stdio.h>
#include <string.h>
...
int length, length2;
char *turkey;
static char *flower= "begonia";
static char *gemstone="ruby ";

length = strlen(flower);
printf("Length = %d\n", length); // prints 'Length = 7'
length2 = strlen(gemstone);

turkey = malloc( length + length2 + 1);
if (turkey) {
  strcpy( turkey, gemstone);
  strcat( turkey, flower);
  printf( "%s\n", turkey); // prints 'ruby begonia'
  free( turkey );
}
```

5

Function: *char ** **strtok** *(char *restrict newstring, const char *restrict delimiters)*

Preliminary: | MT-Unsafe race:strtok | AS-Unsafe | AC-Safe | See POSIX Safety Concepts.

A string can be split into tokens by making a series of calls to the function `strtok`.

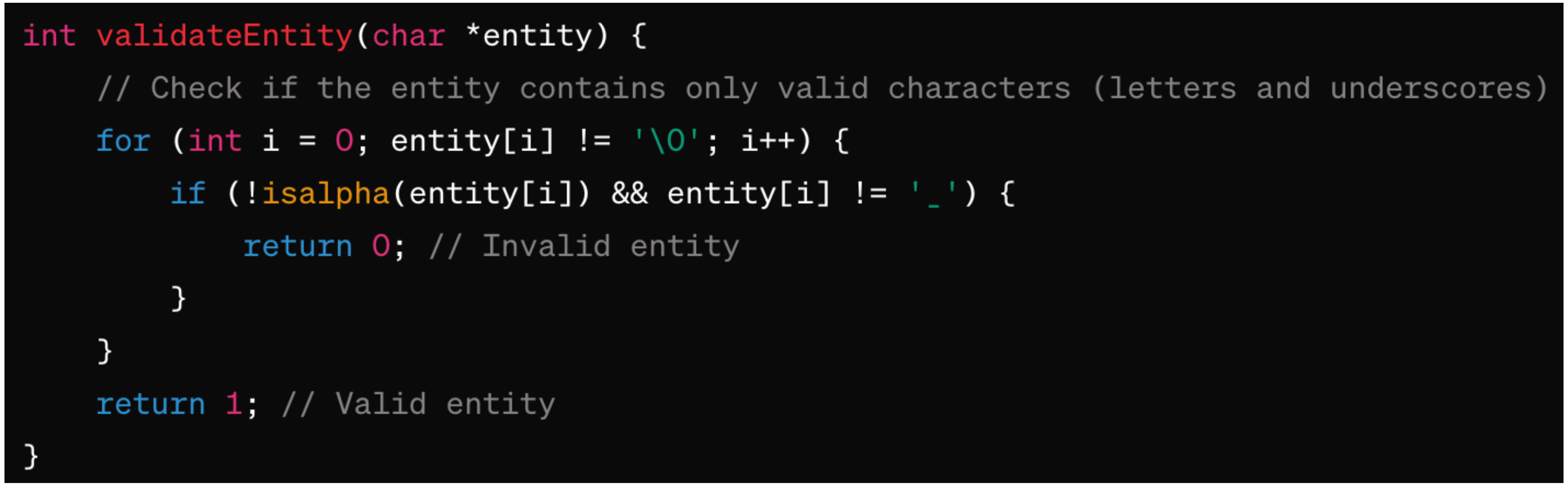
The string to be split up is passed as the *newstring* argument on the first call only. The `strtok` function uses this to set up some internal state information. Subsequent calls to get additional tokens from the same string are indicated by passing a null pointer as the *newstring* argument. Calling `strtok` with another non-null *newstring* argument reinitializes the state information. It is guaranteed that no other library function ever calls `strtok` behind your back (which would mess up this internal state information).

```
#include <stdio.h>
#include <string.h>      /* needed for strtok */

int
main(int argc, char **argv) {
        char text[] = "Cut-down---a----tree++with-a-herring";
        char *t;
        int i;

        t = strtok(text, "-+");
        for (i=0; t != NULL; i++) {
                printf("token %d is \"%s\"\n", i, t);
                t = strtok(NULL, "-+");
        }
}
```

Function: *int* **isalpha** *(int c)*

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See POSIX Safety Concepts.

Returns true if *c* is an alphabetic character (a letter). If `islower` or `isupper` is true of a character, then `isalpha` is also true.

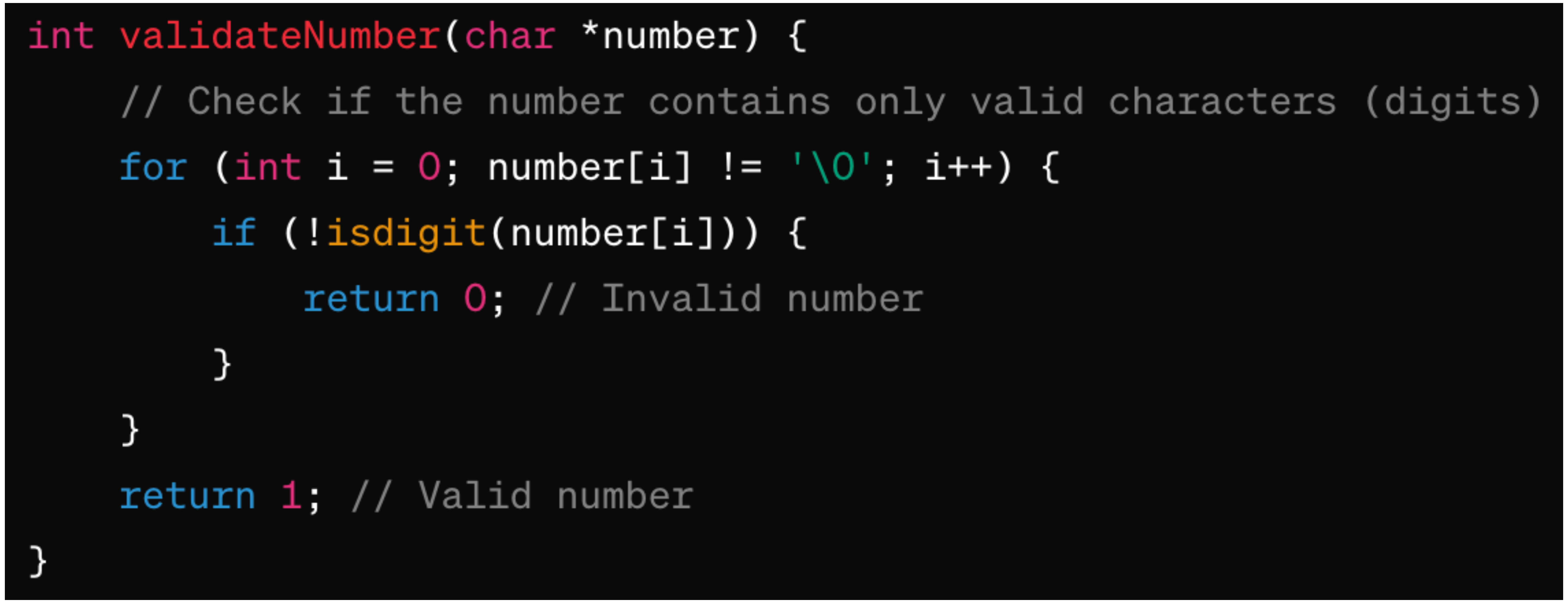


```c
int validateEntity(char *entity) {
    // Check if the entity contains only valid characters (letters and underscores)
    for (int i = 0; entity[i] != '\0'; i++) {
        if (!isalpha(entity[i]) && entity[i] != '_') {
            return 0; // Invalid entity
        }
    }
    return 1; // Valid entity
}
```

Function: *int* **isdigit** *(int c)*

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See POSIX Safety Concepts.

Returns true if *c* is a decimal digit (‘`0`’ through ‘`9`’).



```c
int validateNumber(char *number) {
    // Check if the number contains only valid characters (digits)
    for (int i = 0; number[i] != '\0'; i++) {
        if (!isdigit(number[i])) {
            return 0; // Invalid number
        }
    }
    return 1; // Valid number
}
```